

新・

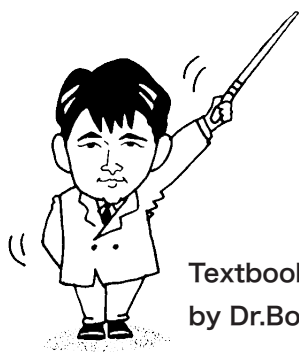
明解

柴田望洋
BohYoh Shibata

Javaで学ぶ アルゴリズムと データ構造 **第3版**

読者特典PDF書籍①

線形リストの応用学習



Textbook **MEIKAI**
by Dr.BohYoh Shibata

 SB Creative

「明解」および「新明解」は、(株)三省堂の登録商標です。

本文中の商品名は、一般に各社の商標または登録商標です。

本文中に、TM、®マークは明記しておりません。

© 2026 本書のプログラムを含むすべての内容は著作権法上の保護を受けています。

著者・発行者の許諾を得ず、無断で複写・複製をすることは禁じられております。

はじめに

『新・明解Javaで学ぶアルゴリズムとデータ構造 第3版』をお買い上げいただき、ありがとうございます。

書籍本文の第8章で学習した**線形リスト**には、“ノードの挿入や削除をデータの移動を伴わずに行える”という特徴がありました。もっとも、ノード用インスタンスの生成と破棄が、挿入や削除のたびに行われることもあり、記憶域の確保・解放に要するコストは決して小さくありません。

データ数の上限が予測できる場合や、プログラムの実行中にデータ数が大きく変化しない場合などは、配列の要素をたくみにやりくりして効率よく運用する手法が適用できます。このPDF書籍では、その手法を学習します。

なお、この手法は、基本情報技術者試験を始めとして、各種資格試験でも問題として頻繁に取りあげられています。また、この手法を応用した考え方が、各種のプログラミング言語の内部で記憶域を管理する際に利用されています。

しっかりと学習を進めていただければ幸いです。

2026年7月

柴田 望洋

本特典書籍は、書籍の第8章『線形リスト』の“続編”という構成です。

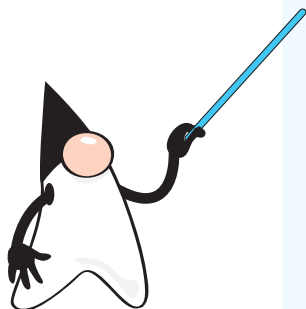
ページ番号は1から始まっていますが、節番号は“8-3節”となっており、図番号とプログラムリストの番号も、書籍に続く番号となっています。

第8章

線形リスト ー発展学習ー

本文で学習した線形リストとは異なる実現法を学習します。予め準備された配列の要素をやりくりする手法です。

- 配列で実現する線形リスト
- フリーリスト



8-3

配列で実現する線形リスト

本節では、各ノードを配列の要素として格納しておき、その要素をたくみにやりくりすることによって実現する線形リストを学習します。

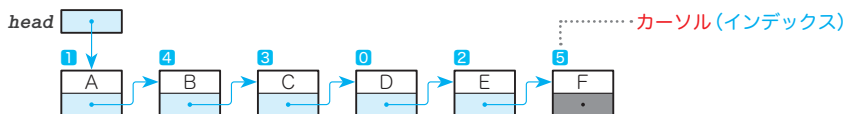
配列で実現する線形リスト

本文で学習した線形リストには、“ノードの挿入や削除をデータの移動を伴わずに行える”という特徴がありました。もっとも、ノード用インスタンスの生成と破棄が、挿入や削除のたびに行われることもあり、記憶域の確保・解放に要するコストは決して小さくありません。

データ数の上限が予測できる場合や、プログラムの実行中にデータ数が大きく変化しない場合などは、配列の要素をたくみにやりくりして効率よく運用する手法が適用できます。

Fig.8-29に示す例で考えていきましょう。図aの線形リストが、各種の情報を加えた上で図bのように配列に格納されています。

a 線形リストの論理的なイメージ



b 配列における物理的な実現

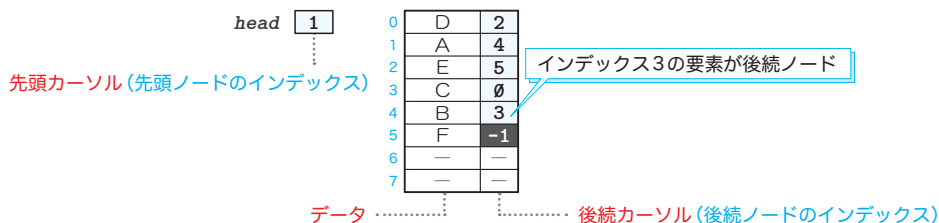


Fig.8-29 カーソルによる線形リスト

各ノードがもつ後続ポインタは、後続ノードへの参照ではなく、後続ノードが格納されている配列内の要素のインデックス (int 型の整数値) です。

整数値のインデックスで表されたポインタをカーソル (cursor) と呼びます。たとえば、ノードBの後続カーソル3は、後続ノードCがインデックス3の要素に入っていることを表します。

なお、末尾ノードの後続カーソルは-1とします。図の例では、末尾ノードFの後続カーソルが-1です。

先頭ノードを表す head もカーソルです。先頭カーソル head の値1は、先頭ノードAがインデックス1の要素に格納されていることを表します。

本手法では、ノードの挿入や削除に伴って、要素を移動する必要がありません。

たとえば、先ほどのリストの先頭にノードGを挿入した後の状態が**Fig.8-30**です。先頭カーソルheadが1から6に変更されて、ノードGの後続カーソルが1に変更されているだけです。

▶ この点は、本文p.267で考えた、“単なる配列で実現した線形リスト”とは、大きく異なります。

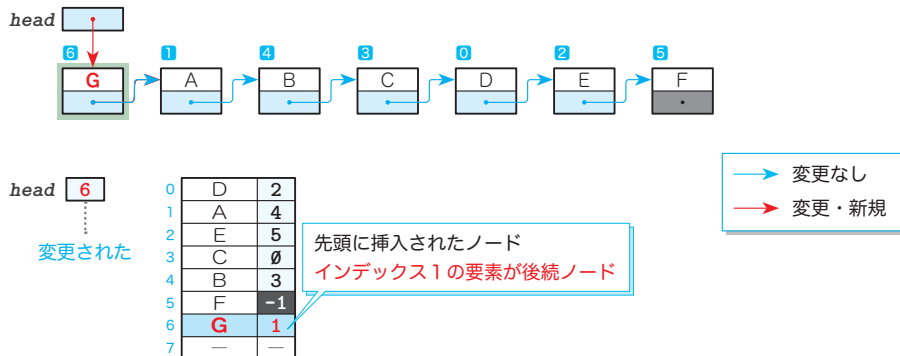


Fig.8-30 先頭へのノードの挿入

以上のアイディアに基づいて実現したクラス `ArrayLinkedList<E>` を **List 8-7** に示します。

▶ 8-1 節の線形リストを **ポインタ版** と呼び、本節の線形リストを **配列カーソル版** と呼びます。

List 8-7 [A]

chap08/ArrayLinkedList.java

// 線形リストクラス（配列カーソル版）

```
import java.util.Comparator;
import java.util.ArrayList;

public class ArrayLinkedList<E> {

    //--- ノード ---//
    class Node<E> {
        E data;           // データ
        int next;         // リストの後続ポインタ
        int dnext;        // フリーリストの後続ポインタ

        //--- 各種セッタ ---//
        void setNext( int next) { this.next = next; }
        void setDnext(int dnext) { this.dnext = dnext; }
        void set(E data, int next) {
            this.data = data; this.next = next;
        }
    }

    private ArrayList<Node<E>> n; // リスト本体
    private int size;           // リストの容量（最大データ数）
    private int max;            // 利用中の末尾レコード
    private int head;           // 先頭ノード
    private int crnt;           // 着目ノード
    private int deleted;        // フリーリストの先頭ノード
    private static final int NULL = -1; // 後続ノードはない／リストは満杯
```



List 8-7 [B]

chap08/ArrayLinkedList.java

```

//--- コンストラクタ ---//
public ArrayLinkedList(int capacity) {
    head = crnt = max = deleted = NULL;
    try {
        n = new ArrayList<>(capacity);
        for (int i = 0; i < capacity; i++)
            n.add(new Node<>());
        size = capacity;
    }
    catch (OutOfMemoryError e) {           // 配列の生成に失敗
        size = 0;
    }
}

//--- 次に挿入するレコードのインデックスを求める ---//
private int getInsertIndex() {
    if (deleted == NULL) {                 // 削除レコードは存在しない
        if (max < size - 1)
            return ++max;                  // 新しいレコードを利用
        else
            return NULL;                   // 容量オーバ
    } else {
        int rec = deleted;                 // フリーリストから
        deleted = n.get(rec).dnext;        // 先頭recを取り出す
        return rec;
    }
}

//--- レコードidxをフリーリストに登録 ---//
private void deleteIndex(int idx) {
    if (deleted == NULL) {                 // 削除レコードは存在しない
        deleted = idx;                    // idxをフリーリストの
        n.set(idx, n.get(idx)).setDnext(NULL); // 先頭に登録
    } else {
        int rec = deleted;                 // idxをフリーリストの
        deleted = idx;                    // 先頭に挿入
        n.set(idx, n.get(idx)).setDnext(rec);
    }
}

//--- ノードを探索 ---//
public E search(E obj, Comparator<? super E> c) {
    int ptr = head;                        // 現在走査中のノード
    while (ptr != NULL) {
        if (c.compare(obj, n.get(ptr).data) == 0) {
            crnt = ptr;
            return n.get(ptr).data;        // 探索成功
        }
        ptr = n.get(ptr).next;             // 後続ノードに着目
    }
    return null;                           // 探索失敗
}

//--- 先頭にノードを挿入 ---//
public void addFirst(E obj) {
    int ptr = head;                        // 挿入前の先頭ノード
    int rec = getInsertIndex();
    if (rec != NULL) {
        head = crnt = rec;                // 第recレコードに挿入
        n.set(head, n.get(head)).set(obj, ptr);
    }
}

```

```

//--- 末尾にノードを挿入 ---//
public void addLast(E obj) {
    if (head == NULL) {
        addFirst(obj);
    }
    else {
        int ptr = head;
        while (n.get(ptr).next != NULL)
            ptr = n.get(ptr).next;
        int rec = getInsertIndex();
        if (rec != NULL) {
            n.set(ptr, n.get(ptr)).setNext(crnt = rec);
            n.set(rec, n.get(rec)).set(obj, NULL);
        }
    }
}

//--- 先頭ノードを削除 ---//
public void removeFirst() {
    if (head != NULL) {
        int ptr = n.get(head).next;
        deleteIndex(head);
        head = crnt = ptr;
    }
}

//--- 末尾ノードを削除 ---//
public void removeLast() {
    if (head != NULL) {
        if (n.get(head).next == NULL)
            removeFirst();
        else {
            int ptr = head;
            int pre = head;
            while (n.get(ptr).next != NULL) {
                pre = ptr;
                ptr = n.get(ptr).next;
            }
            n.set(pre, n.get(pre)).setNext(NULL);
            deleteIndex(ptr);
            crnt = pre;
        }
    }
}

//--- 着目ノードを削除 ---//
public void removeCurrentNode() {
    if (head != NULL) {
        if (crnt == head)
            removeFirst();
        else {
            int ptr = head;
            while (n.get(ptr).next != crnt) {
                ptr = n.get(ptr).next;
            }
            if (ptr == NULL) return;
            n.set(ptr, n.get(ptr)).setNext(n.get(crnt).next);
            deleteIndex(crnt);
            crnt = ptr;
        }
    }
}

```

// リストが空であれば
// 先頭に挿入

// 第recレコードに挿入

// リストが空でなければ

// ノードが一つだけであれば
// 先頭ノードを削除

// 走査中のノード
// 走査中のノードの先行ノード

// preは削除後の末尾ノード

// crntが先頭ノードであれば
// 先頭ノードを削除

// crntはリスト上に存在しない

8-3

配列で実現する線形リスト



```

//--- 全ノードを削除 ---//
public void clear() {
    while (head != NULL)
        removeFirst();
    crnt = NULL;
}

//--- 着目ノードを一つ後方に進める ---//
public boolean next() {
    if (crnt == NULL || n.get(crnt).next == NULL)
        return false;
    crnt = n.get(crnt).next;
    return true;
}

//--- 着目ノードを表示 ---//
public void printCurrentNode() {
    if (crnt == NULL)
        IO.println("着目ノードはありません。");
    else
        IO.println(n.get(crnt).data);
}

//--- 全ノードを表示 ---//
public void dump() {
    int ptr = head;
    while (ptr != NULL) {
        IO.println(n.get(ptr).data);
        ptr = n.get(ptr).next;
    }
}
}

```

配列内の空き要素

クラス内の各メソッドは、8-1 節のポインタ版のプログラムと、おおむね一対一に対応していますので、大きく異なる“**削除されたノードの管理**”に絞って学習していきます。

まずは、**Fig.8-31**を例に、ノードの挿入と削除について考えていきましょう。

- a** 線形リストに4個のノードが{A⇒B⇒C⇒D}の順序で並んでいて、右図の配列に格納されている状態です。
- b** 線形リストの先頭にノードEを挿入した後の状態です。インデックス4の位置にノードEが格納されています。

挿入されたノードの格納場所は、“配列の(物理的な)末尾側のインデックスの位置”であって、決して“線形リストとしての(論理的な)末尾”ではありません。

当然のことですが、リスト上で第*n*番目に位置するノードが、配列のインデックス*n*の要素に格納されるとは限りません。配列上の物理的な位置関係と、線形リスト上の論理的な順序関係は必ずしも一致しないからです。

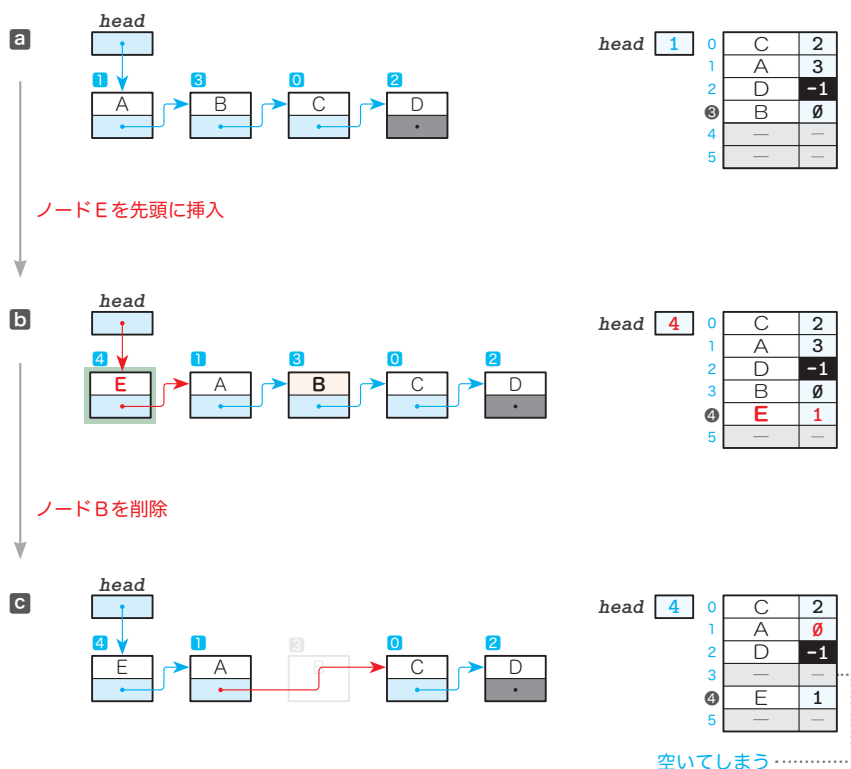


Fig.8-31 線形リストに対するノードの挿入と削除

これ以降、リスト上の位置と区別するために、配列中のインデックス n の要素に格納されているノードを**第 n レコード**と呼びます。

▶ リストの先頭に挿入されたノードEは、第4レコードに格納されたわけです。

c 図**b**の状態から、先頭から3番目に位置するノードBを削除した後の状態です。それまでノードBのデータが格納されていた第3レコードが空きます。

*

もし削除が何度も繰り返されると、配列の中は空きレコードだらけとなってしまいます。

削除されるレコードが高々一つであれば、そのインデックスを何らかの変数に入れておいて管理することによって、そのレコードを容易に再利用できます。

しかし、実際には複数のレコードが削除されるわけですから、そう単純にはいきません。

フリーリスト

削除されたレコード群の管理に導入されているのが、その並びを格納するための、線形リスト構造で実現された**フリーリスト** (free list) です。

データそのものの順序を表す線形リストと、フリーリスト用の線形リストとの組合せで管理されるため、ノードクラス `Node<E>` と線形リストクラス `ArrayLinkedList<E>` の両方に、ポインタ版の各クラスに対してフィールドが追加されています。

■ ノードクラス `Node<E>` に追加されたフィールド

■ `dnext`

フリーリストの後続カーソル (フリーリスト上における後続ノードを参照するカーソル)。

■ 線形リストクラス `ArrayLinkedList<E>` に追加されたフィールド

■ `deleted`

フリーリストの先頭カーソル (フリーリストの先頭ノードを指すカーソル)。

■ `max`

配列中の最も末尾側の位置に格納されているノードのレコード番号。

▶ 前ページの **Fig.8-31** の●内の値が `max` です (この値は `3⇒4⇒4` と変化していました)。

右ページの **Fig.8-32** を見ながら、ノードの挿入と削除に伴うフリーリストの変化を理解しましょう。

- a** 線形リストに5個のノードが `{A⇒B⇒C⇒D⇒E}` の順序で並んでいます。 `max` は7であり、第8レコード以降が未使用の状態です。また、レコード1と3と5が削除済みの空きレコードであって、フリーリストは `{3⇒1⇒5}` となっています。

▶ 図にも示すように、本来のデータの並びとしての線形リストに加えて、削除されたレコード群を管理するための線形リスト=フリーリストが存在します。

フリーリストの先頭カーソル (先頭ノードのインデックス) 3 を格納するのが、線形リストクラス `ArrayLinkedList` のフィールド `deleted` です。

- b** 線形リストの末尾にノードFを挿入した後の状態です。ノードFの格納先レコードは、フリーリスト `{3⇒1⇒5}` の先頭の3です。第3レコードが空き状態ではなくなったため、フリーリストから3を削除して `{1⇒5}` に更新します。

このように、フリーリスト上に空きレコードが登録されている限り、“未使用レコード (すなわち第 `max` レコード以降のレコード) を求めて `max` を増やして、その位置にデータを格納する”といったことは行いません。そのため、 `max` の値は7のままです。

- c** ノードDを削除した後の状態です。第7レコードに格納されているデータが削除されるため、その7をフリーリストの先頭に登録します。

その結果、フリーリストは、 `{1⇒5}` から `{7⇒1⇒5}` へと更新されます。

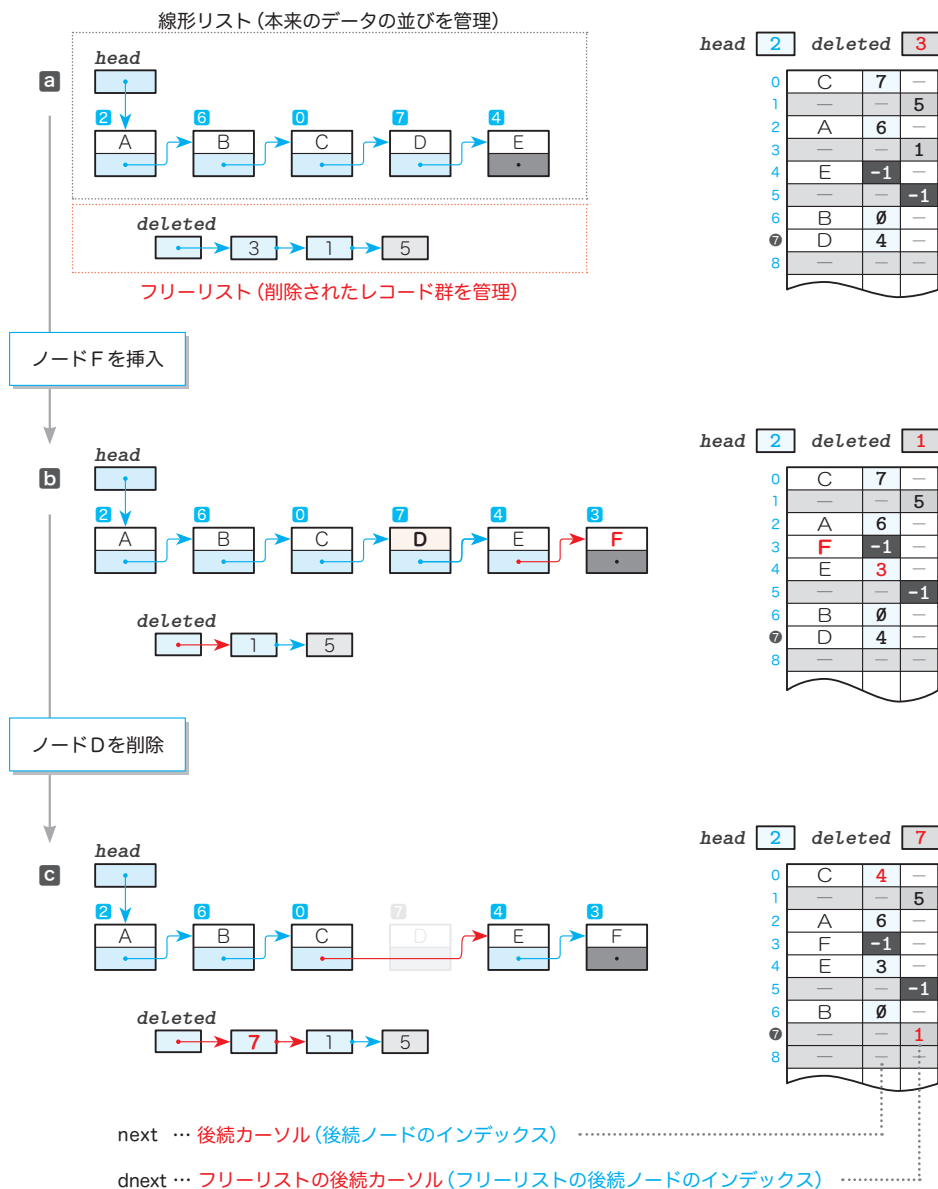


Fig.8-32 ノードの挿入と削除に伴うフリーリストの変化

▶ 削除された記録をフリーリストに登録するのがメソッド `deleteIndex` です。

また、ノードの挿入時に、格納する記録番号を決定するのがメソッド `getInsertIndex` です。

図 8-32 では削除記録が存在するため、フリーリストの先頭に登録されている記録に挿入されたノードを格納しています。なお、削除記録が存在せずフリーリストが空であれば、`max`を増やして配列末尾側の未使用記録を利用します。

□ 配列カーソル版の線形リストを利用するプログラム例

配列カーソル版の線形リストクラス `ArrayLinkedList<E>` を利用するプログラムを作ります。ポインタ版と同様に、各ノードのデータとしては会員クラス `Member` を使いましょう。

プログラムを **List 8-8**、**List 8-9**、**List 8-10** に示します。

List 8-8

chap08/Member.java

```
// 会員クラス

import java.util.Comparator;

//--- 会員クラス (会員番号+氏名) ---//
public class Member {
    static final int NO    = 1;    // 番号を読み込むか?
    static final int NAME  = 2;    // 氏名を読み込むか?

    private Integer no = 0;    // 会員番号 (キー値)
    private String  name = ""; // 氏名

    //--- キー値 ---//
    Integer keyCode() {
        return no;
    }

    //--- 文字列表現 ---//
    public String toString() {
        return "{" + no + ", " + name + "}";
    }

    //--- 番号・氏名の読み込み ---//
    void read(String guide, int sw) {
        IO.println(guide + "するデータを入力せよ。");

        if ((sw & NO) == NO)
            no = Integer.parseInt(IO.readLine("番号 : "));
        if ((sw & NAME) == NAME)
            name = IO.readLine("氏名 : ");
    }

    //--- 会員番号による順序付けを行うコンパレータ ---//
    public static final Comparator<Member> NO_ORDER = new NoOrderComparator();

    private static class NoOrderComparator implements Comparator<Member> {
        public int compare(Member m1, Member m2) {
            return m1.no.compareTo(m2.no);
        }
    }

    //--- 氏名による順序付けを行うコンパレータ ---//
    public static final Comparator<Member> NAME_ORDER = new NameOrderComparator();

    private static class NameOrderComparator implements Comparator<Member> {
        public int compare(Member m1, Member m2) {
            return m1.name.compareTo(m2.name);
        }
    }
}
```

▶ 会員クラス `Member` は、第3章のハッシュ、第8章の線形リスト、第9章の2分探索木で共通です。ここでは、クラス定義の全体を示しています。

※本文の **List 3-10** (p.104) ではコンパレータの定義の提示を省略し、**List 8-2** (p.281) ではメソッド `keyCode`、`toString` メソッド、メソッド `read` のコードの提示を省略していました。

List 8-9

chap08/Menu.java

```

//--- 線形リスト用メニュー ---//
public enum Menu {
    TERMINATE( "終了"),
    ADD_FIRST( "先頭にノードを挿入"),
    ADD_LAST( "末尾にノードを挿入"),
    RMV_FIRST( "先頭ノードを削除"),
    RMV_LAST( "末尾ノードを削除"),
    RMV_CRNT( "着目ノードを削除"),
    CLEAR( "全ノードを削除"),
    SEARCH_NO( "番号で探索"),
    SEARCH_NAME("氏名で探索"),
    NEXT( "着目ノードを進める"),
    PRINT_CRNT( "着目ノードを表示"),
    DUMP( "全ノードを表示");

    private final String name;          // 文字列表現

    //--- コンストラクタ ---//
    private Menu(String string) {
        name = string;
    }

    //--- 序数がidxのメニューを返す ---//
    private static Menu menuAt(int idx) {
        for (Menu m : Menu.values())
            if (m.ordinal() == idx)
                return m;
        return null;
    }

    //--- メニュー選択 ---//
    public static Menu selectMenu() {
        int key;
        do {
            for (int i = 1; i < Menu.values().length; i++) {
                Menu m = Menu.values()[i];
                IO.print("(%d) %s ".formatted(m.ordinal(), m.name));
                if (i % 3 == 0) IO.println();
            }
            IO.print("(%d) %s".formatted(TERMINATE.ordinal(), TERMINATE.name));
            IO.print(" : ");
            key = Integer.parseInt(IO.readLine());
        } while (key < Menu.TERMINATE.ordinal() || key > Menu.DUMP.ordinal());

        return menuAt(key);
    }
}

```

8-3

配列で実現する線形リスト

- ▶ ここに示すメニュークラスMenuは、本文のList 8-3 (p.281)と同じものです。

List 8-10

chap08/ArrayLinkedListTester.java

// 線形リストクラスArrayLinkedList<E>の利用例

要 : ArrayLinkedList・Menu・Member

```

void main() {
    Menu menu;                // メニュー
    Member data;              // 追加用データ参照
    Member ptr;               // 探索用データ参照
    Member temp = new Member(); // 読み込み用データ

    ArrayLinkedList<Member> list = new ArrayLinkedList<>(100);

    do {
        switch (menu = Menu.selectMenu()) {
            case ADD_FIRST -> {                // 先頭にノードを挿入
                data = new Member();
                data.read("先頭に挿入", Member.NO | Member.NAME);
                list.addFirst(data);
            }
            case ADD_LAST -> {                // 末尾にノードを挿入
                data = new Member();
                data.read("末尾に挿入", Member.NO | Member.NAME);
                list.addLast(data);
            }
            case RMV_FIRST -> list.removeFirst(); // 先頭ノードを削除
            case RMV_LAST -> list.removeLast();  // 末尾ノードを削除
            case RMV_CRNT -> list.removeCurrentNode(); // 着目ノードを削除
            case SEARCH_NO -> {                // 会員番号で探索
                temp.read("探索", Member.NO);
                ptr = list.search(temp, Member.NO_ORDER);
                if (ptr == null)
                    IO.println("その番号のデータはありません。");
                else
                    IO.println("探索成功 : " + ptr);
            }
            case SEARCH_NAME -> {            // 氏名で探索
                temp.read("探索", Member.NAME);
                ptr = list.search(temp, Member.NAME_ORDER);
                if (ptr == null)
                    IO.println("その氏名のデータはありません。");
                else
                    IO.println("探索成功 : " + ptr);
            }
            case NEXT -> list.next();        // 着目ノードを後方に進める
            case PRINT_CRNT -> list.printCurrentNode(); // 着目ノードのデータを表示
            case DUMP -> list.dump();        // 全データをリスト順に表示
            case CLEAR -> list.clear();      // 全ノードを削除
        }
    } while (menu != Menu.TERMINATE);
}

```

- 本プログラムが、配列カーソル版線形リストの利用例のメインとなるプログラムです。
 実行に際しては、同一フォルダ上に "ArrayLinkedList.class" と "Menu.class" と "Member.class" が必要です。

実行例

- (1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 1

先頭に挿入するデータを入力せよ。

番号: 1

{ ① 赤尾 } を先頭に挿入

氏名: 赤尾

- (1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 2

末尾に挿入するデータを入力せよ。

番号: 5

{ ⑤ 武田 } を末尾に挿入

氏名: 武田

- (1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 1

先頭に挿入するデータを入力せよ。

番号: 10

{ ⑩ 小野 } を先頭に挿入

氏名: 小野

- (1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 2

末尾に挿入するデータを入力せよ。

番号: 12

{ ⑫ 鈴木 } を末尾に挿入

氏名: 鈴木

- (1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 1

先頭に挿入するデータを入力せよ。

番号: 14

{ ⑭ 神崎 } を先頭に挿入

氏名: 神崎

- (1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 4

末尾の { ⑫ 鈴木 } を削除

- (1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 8

探索するデータを入力せよ。

氏名: 鈴木

{ 鈴木 } を探索 / 失敗

その氏名のデータはありません。

- (1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 7

探索するデータを入力せよ。

番号: 10

{ ⑩ } を探索 / 成功

探索成功: { ⑩, 小野 }

- (1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 10

(10) 小野

着目ノードは { ⑩ 小野 }

8-3

配列で実現する線形リスト



実行例（前ページの続き）

(1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 11

{14, 神崎}

{10, 小野}

{1, 赤尾}

{5, 武田}

全ノードを順に表示

(1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 7

探索するデータを入力せよ。

番号: 1

探索成功: {1, 赤尾}

{1}を探索/成功

(1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 5

着目ノードを削除

(1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 3

先頭ノードを削除

(1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 11

{10, 小野}

{5, 武田}

全ノードを順に表示

(1) 先頭にノードを挿入 (2) 末尾にノードを挿入 (3) 先頭ノードを削除
 (4) 末尾ノードを削除 (5) 着目ノードを削除 (6) 全ノードを削除
 (7) 番号で探索 (8) 氏名で探索 (9) 着目ノードを進める
 (10) 着目ノードを表示 (11) 全ノードを表示 (0) 終了: 0

制 作 … 柴田 望洋
装 丁 … 柴田 望洋
編 集 … 福井 荘介

新・明解Javaで学ぶアルゴリズムとデータ構造 第3版

読者特典PDF書籍① 線形リストの応用学習

2026年9月10日 初版発行

著 者 … 柴田 望洋
発行者 … 出井 貴完
発行所 … S Bクリエイティブ株式会社
〒105-0001 東京都港区虎ノ門2-2-1
<https://www.sbcr.jp/>